Delay Modeling and Static Timing Verification

Prof. Kurt Keutzer

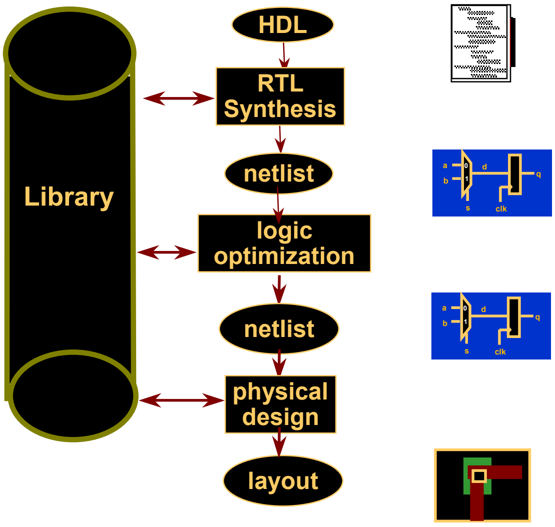
Michael Orshansky

EECS

University of California

Berkeley, CA

RTL Synthesis Flow



Library

HDL

RTL Synthesis

netlist

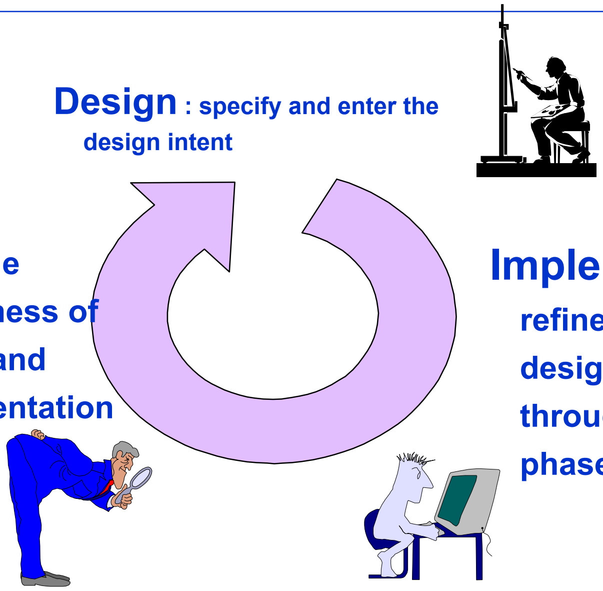
logic optimization

netlist

physical design

layout

Design Process



Design : specify and enter the design intent

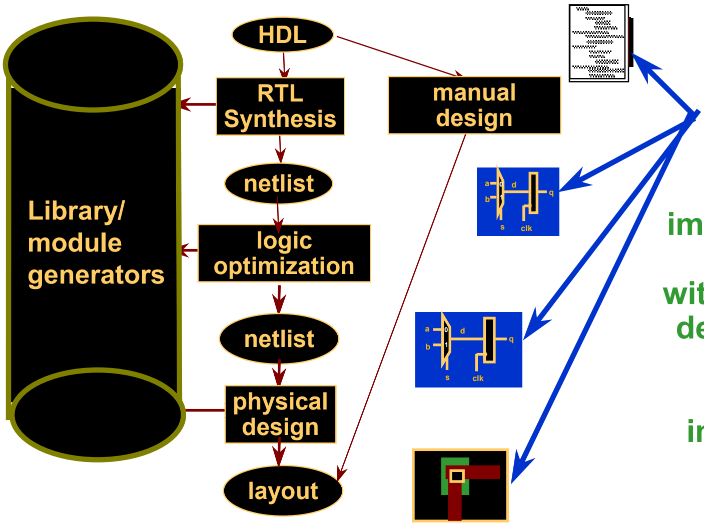
Verify:

verify the correctness of design and implementation

Implement:

refine the design through all phases

Implementation Verification



HDL

RTL Synthesis

netlist

logic optimization

netlist

physical design

layout

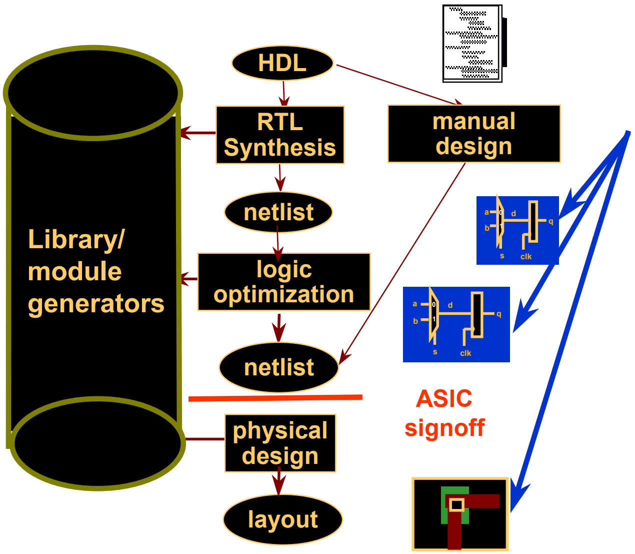
manual design

Library/ module generators

Is the implementation consistent with the original design intent?

Is what I implemented what I wanted?

Implementation verification for ASIC's

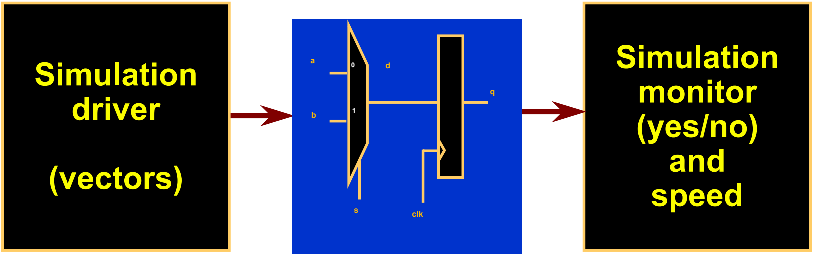


Apply gate-level simulation ("the golden simulator") at each step to verify

functionality:
- 0-1 behavior on regression test set

and timing:
- maximum delay of circuit across critical paths

Software Simulation



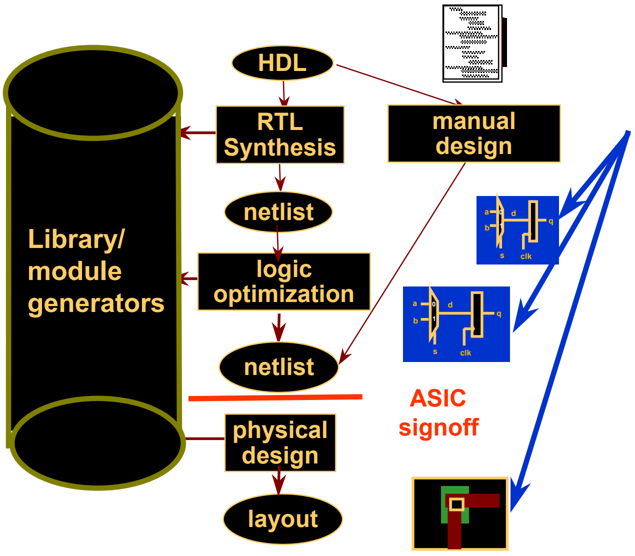
Advantages of gate-level simulation

- verifies timing and functionality simultaneously
- approach well understood by designers

Disadvantages of gate-level simulation

- computationally intensive - only 1 - 10 clock cycles of 100K gate design per 1 CPU second
- incomplete - results only as good as your vector set - easy to overlook incorrect timing/behavior

Alternative - Static Sign-off



Use static analysis techniques to verify:

functionality:

- *formal equivalence-checking techniques – we'll talk about this later*

and timing:

- use static timing analysis

Different Roles of Timing Analysis

Optimization is only relevant when there exists an objective function

For many circuits the primary objective function is speed

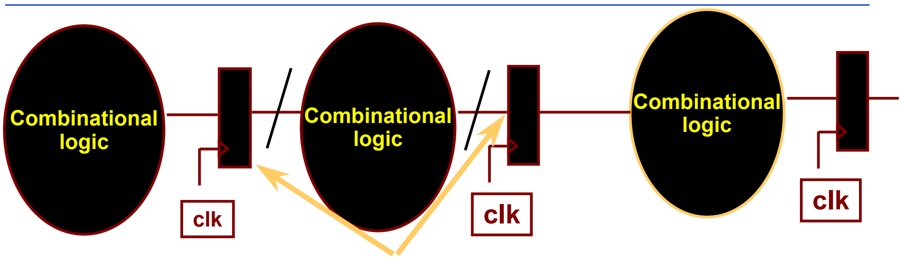
Timing verification

- Before fabrication, ensure a chip meets its timing requirements

Timing-driven optimization – give fast accurate timing information to guide tools as they evolve the chip

- Logic synthesis
- Placement
- Routing

Approach of Static Timing Verification



- determine fastest permissible clock speed (e.g. 100MHz) by determining delay (including set-up and hold time) of longest path from register to register (e.g. 10ns.)

- largely eliminates need for gate-level simulation to verify the delay of the circuit

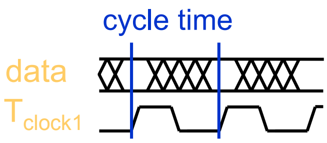
Cycle Time - Critical Path Delay

Cycle time (T) cannot be smaller than longest path delay (T_{max})

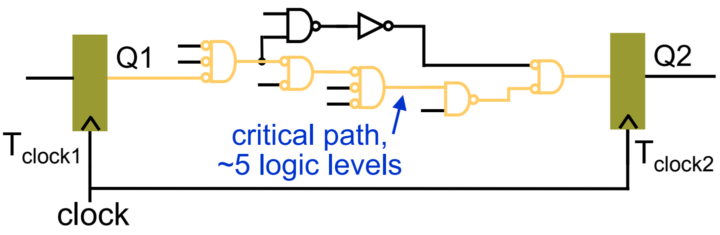
Longest (critical) path delay is a function of:

Total gate, wire delays

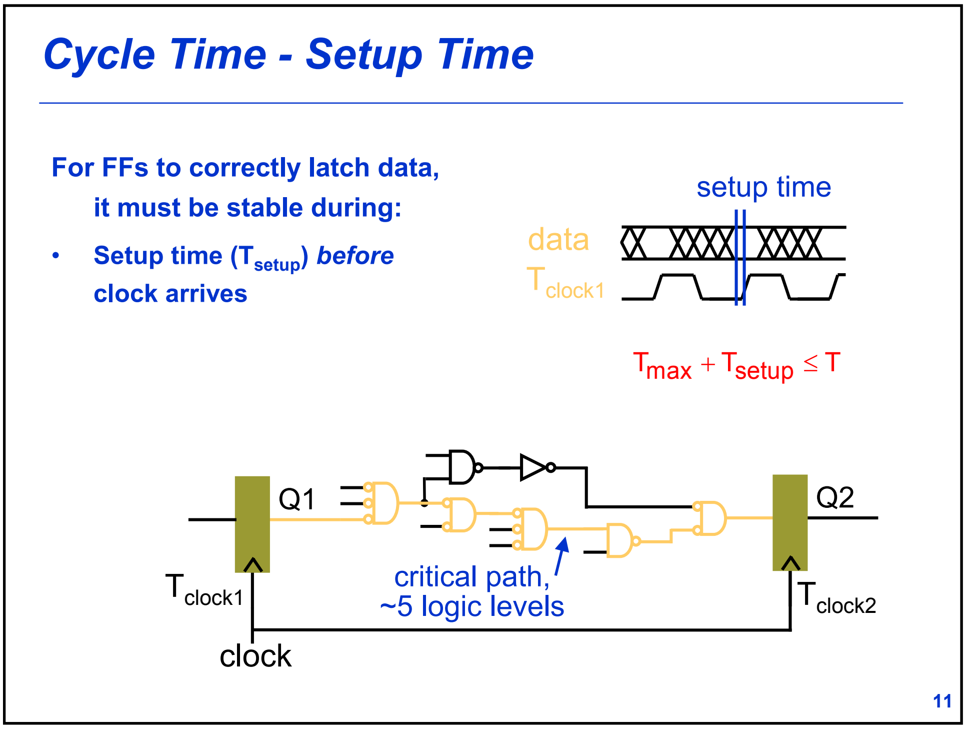
- logic levels



$T_{max} \leq T$



critical path, ~5 logic levels

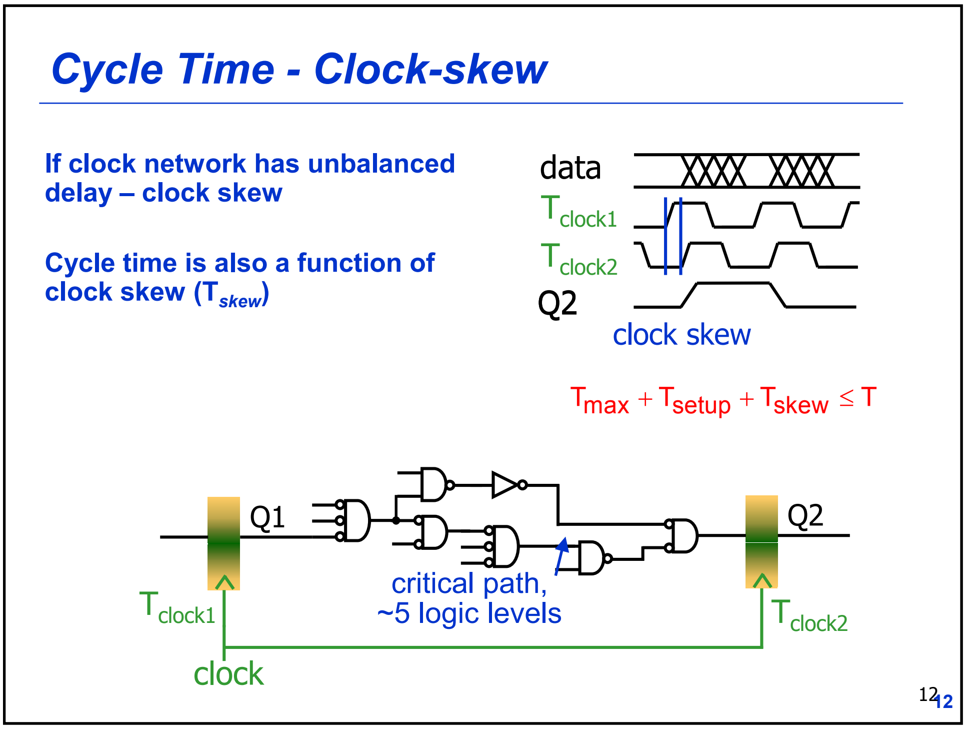


Cycle Time - Setup Time

For FFs to correctly latch data,
it must be stable during:

- Setup time (T_{setup}) *before* clock arrives

$T_{max} + T_{setup} \leq T$

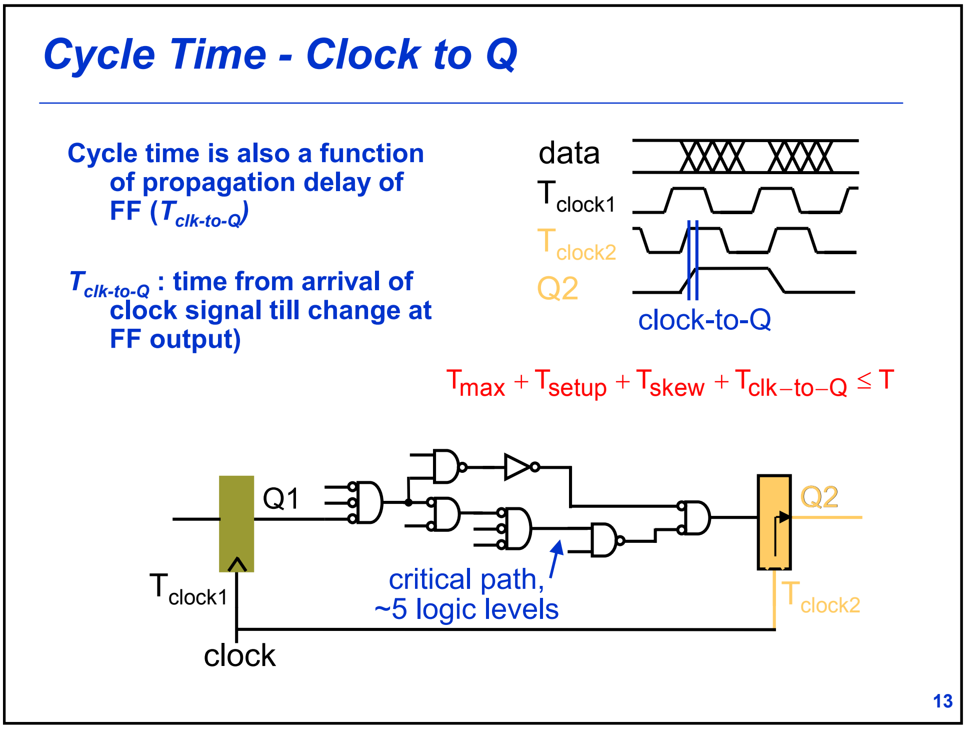


Cycle Time - Clock-skew

If clock network has unbalanced delay – clock skew

Cycle time is also a function of clock skew (T_{skew})

$T_{max} + T_{setup} + T_{skew} \leq T$



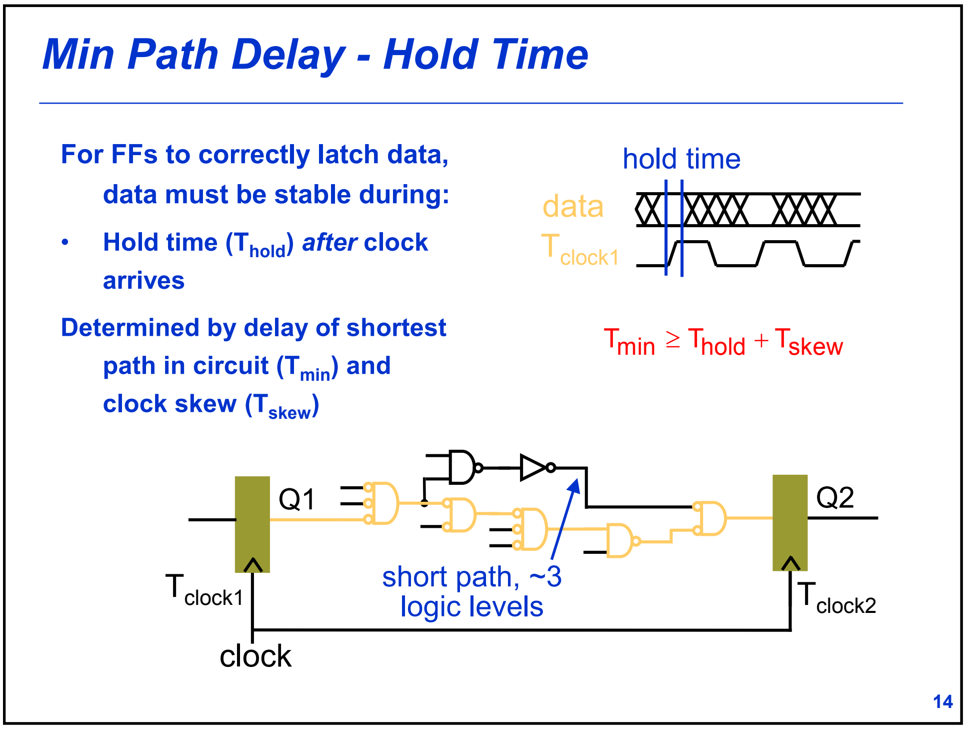
Cycle Time - Clock to Q

Cycle time is also a function of propagation delay of FF (*$T_{clk-to-Q}$*)

***$T_{clk-to-Q}$* : time from arrival of clock signal till change at FF output)**

$$T_{max} + T_{setup} + T_{skew} + T_{clk-to-Q} \le T$$

critical path, ~5 logic levels



Min Path Delay - Hold Time

For FFs to correctly latch data, data must be stable during:

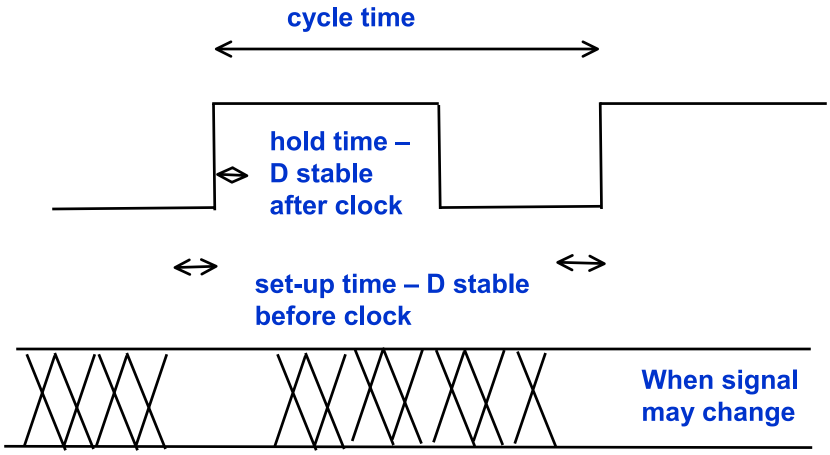
- **Hold time (T_{hold}) *after* clock arrives**

Determined by delay of shortest path in circuit (T_{min}) and clock skew (T_{skew})

$$T_{min} \ge T_{hold} + T_{skew}$$

short path, ~3 logic levels

One more time



cycle time

hold time –
D stable
after clock

set-up time – D stable
before clock

When signal
may change

Example of a single phase clock

Elements of Timing Verification

To verify circuit timing need

- Accurate delay calculation
- Timing analysis engine

Delay calculation

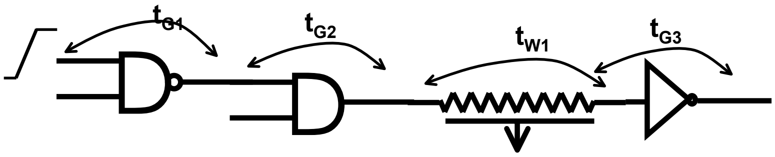
- Delay numbers for gates
- Delay numbers for wires

Timing analysis engine

- Circuit path analysis
- Integrating clock network and FF/latches

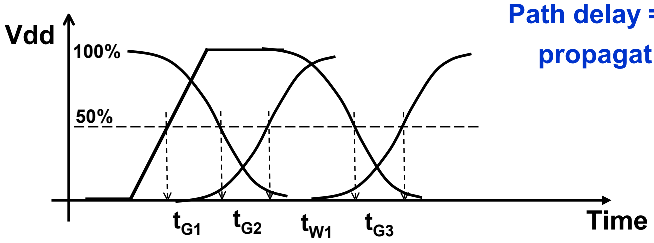
Delay Modeling and Delay Computation

Single path delay computation



To simulate complex circuits, need accurate models of

- Gate delay
- Interconnect delay



Path delay = sum of 50% propagation delays

Gate Delay Modeling Requirements

Fast delay evaluation

- To enable full chip simulation
- Analytical models and look-up tables

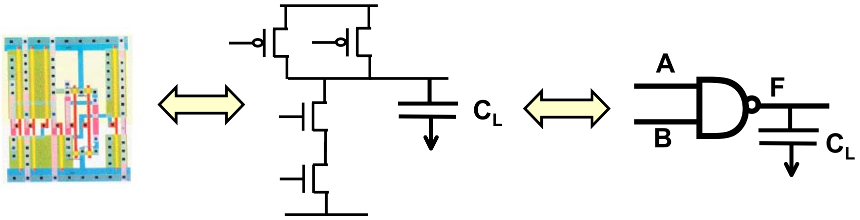
Conservative delay models

- STA determines longest path delay under *all possible* conditions

To enable fast tractable computation, have to give up on many modeling details

- Input pattern dependencies
- Complex dynamic behavior is captured through tables

Gate Timing Characterization



"Extract" exact transistor characteristics from layout

- Transistor width, length, junction area and perimeter
- Local wire length and inter-wire distance

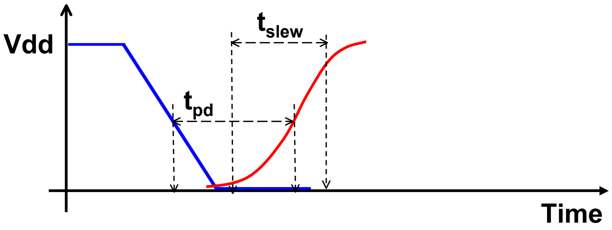
Compute all transistor and wire capacitances

Cell Timing Characterization

Delay tables generated using a detailed transistor-level circuit simulator SPICE (differential-equations solver)

For a number of different input slews and load capacitances simulate the circuit of the cell

- Propagation time (50% Vdd at input to 50% at output)
- Output slew (10% Vdd at output to 90% Vdd at output)

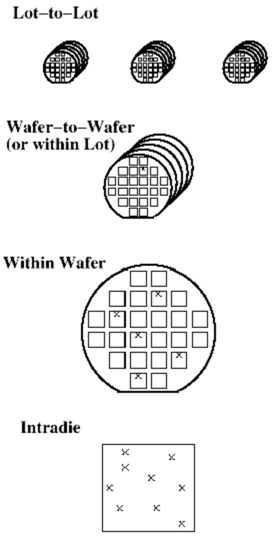


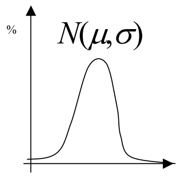
Scope of Variation

Inter-Die Variation

Intra-Die Variation

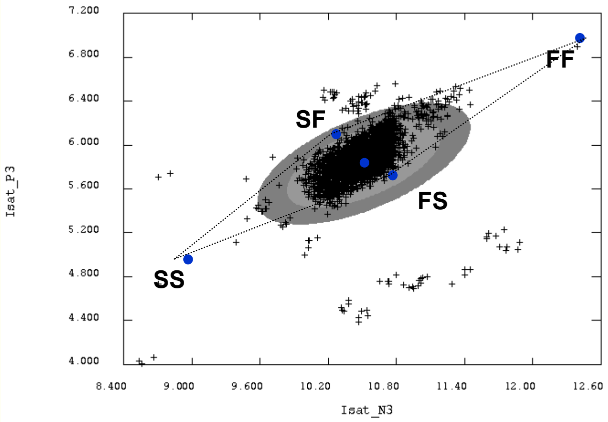
- As an ASIC vendor we must ensure that our chips work across this range of variation
- This is a *very restrictive condition and will be responsible for many conservative assumptions throughout timing analysis*





Device Worst Case Model

Comparison Corner vs Real Data



● Corners from SPICE document

FF: Fast NMOS
&
Fast PMOS

SF: Slow NMOS
&
Fast PMOS

3 corner Model:
TT, SS, FF

5 corner model:
all

Must use worst case assumptions for Slow/set-up and Fast/hold analysis

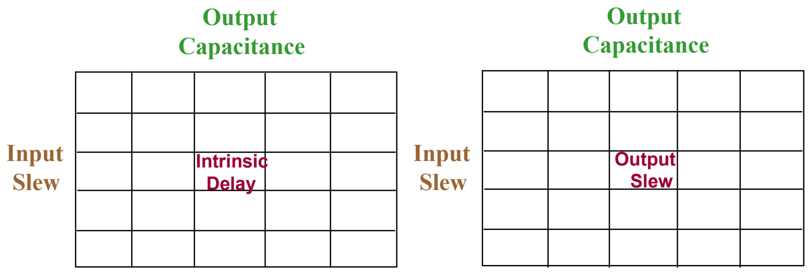
How Is Gate Delay Computed? Non-linear effects reflected in tables

$D_G = f(C_L, S_{in})$ and $S_{out} = f(C_L, S_{in})$

- Non-linear

Interpolate between table entries

Interpolation error is usually below 10% of SPICE



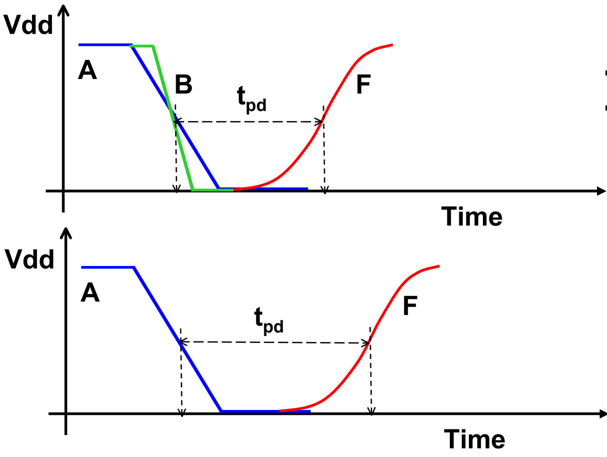
Delay at the gate

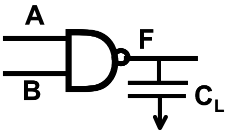
Resulting waveform

Conservatism of Gate Delay Modeling

True gate delay depends on input arrival time patterns

- STA will assume that only 1 input is switching
- Will use worst slope among several inputs





Elements of Timing Verification

To verify circuit timing need

- Accurate delay calculation
- Timing analysis engine

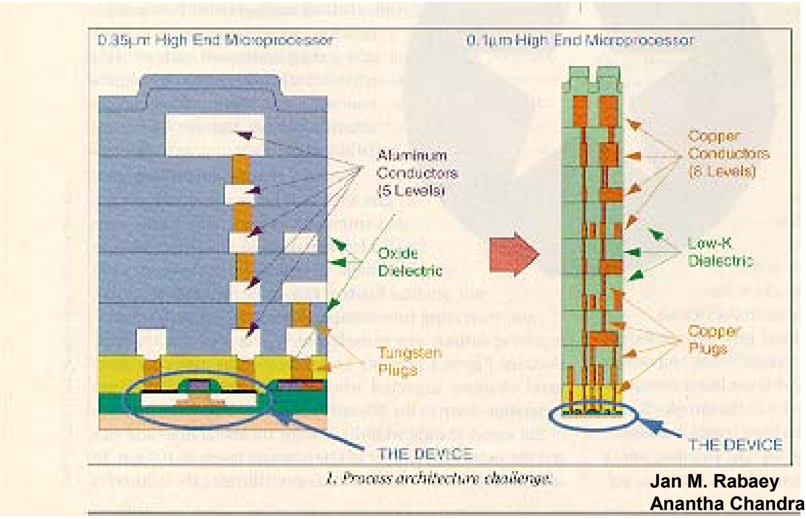
Delay calculation

- Delay numbers for gates
- Delay numbers for wires

Timing analysis engine

- Circuit path analysis
- Integrating clock network and FF/latches

Interconnect Impact on Chip


0.35μm High End Microprocessor
0.1μm High End Microprocessor
Aluminum Conductors (5 Levels)
Oxide Dielectric
Tungsten Plugs
THE DEVICE
Copper Conductors (6 Levels)
Low-K Dielectric
Copper Plugs
THE DEVICE

1. Process architecture challenge.

Jan M. Rabaey
Anantha Chandrakasan
Borivoje Nikolic

Wire-Dominated Chips

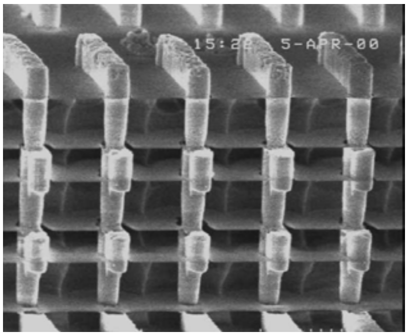
Wiring requirements grow dramatically

Number of interconnect layers (6-8)

Interconnect effects

- Large RC and RLC delays
- Inter-wire coupling

Interconnect becomes a dominant factor in limiting chip performance

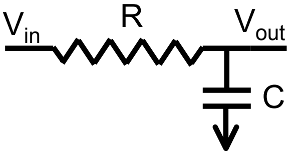


TSMC Copper Process

Wire Delay Modeling

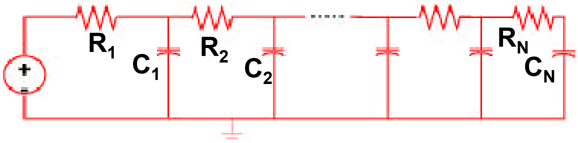
Lumped RC model

- Simple: R – total resistance, C – total capacitance
- Pessimistic and inaccurate
- Spurious oscillations



Distributed RC model

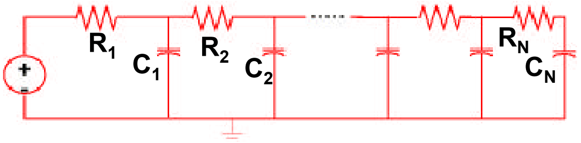
- Required for longer interconnect lines
- Exact solution requires solving "diffusion equation"
- No closed-form solution - approximations



Wire Delay Modeling: Elmore Constant

Elmore Delay Constant

- Dominant time constant for step input
- Works for branch-less distributed RC networks
- No floating caps, grounded resistors



$$\tau_N = \sum_{i=1}^{N} C_i \sum_{j=1}^{i} R_j = C_1R_1 + C_2(R_1 + R_2) + \ldots + C_i(R_1 + R_2 + \ldots + R_i)$$

Wire Delay Modeling: AWE

AWE (Asymptotic Waveform Evaluation) is a qth-order extension of the Elmore delay for general RLC circuits

A generalized approach to linear RLC network response approximations

- Floating cap, grounded resistors, inductors
- Non-zero input transition times
- Initial conditions

Produces a reduced qth order model of transient response

- Trade-off between model order (complexity) and model accuracy

The q unknown time- constants, or poles, of the circuit, are obtained through a moment matching technique (a Padé approximation)

A first-order AWE approximation reduces to RC tree methods

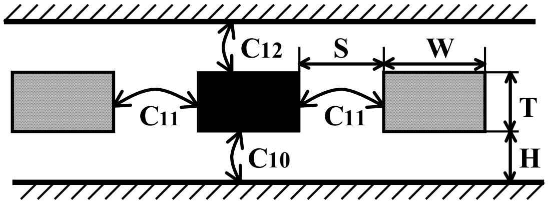
Constructing RC Network

Resistance estimation (extraction) is easy

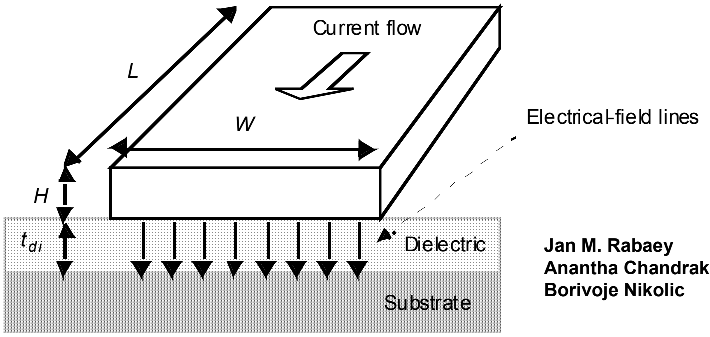
$$R_W = \rho \frac{L_W}{W \cdot T}$$

Capacitance estimation (extraction) is difficult

- **Analytically**
- **Using electromagnetic 3D simulators / extractors**



Capacitance: The Parallel Plate Model

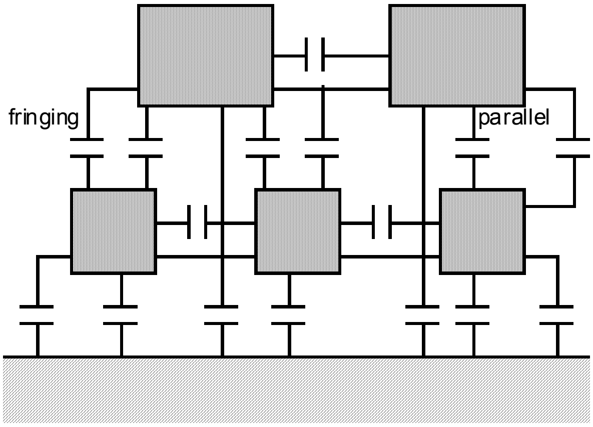


Jan M. Rabaey
Anantha Chandrakasan
Borivoje Nikolic

$$c_{int} = \frac{\varepsilon_{di}}{t_{di}} WL$$

$$S_{Cwire} = \frac{S}{S \cdot S_L} = \frac{1}{S_L}$$

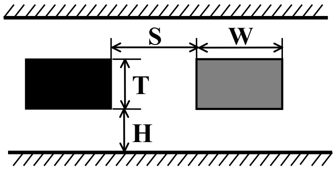
Interwire Capacitance



Deriving Capacitance Information

Wire delays determined by layout

- Wire-to-wire spacing
- Wire-length



Problem: wire-to-wire capacitance and wire-length not known until placement and routing of all circuit components

Two modes of evaluating wire delay

- Post P&R: 2, 2&1/2 or 3-D interconnect parasitic extraction (e.g. capacitance and resistance), distributed RC delay models
- Synthesis: tabular data based on prior chips

Wire Load Models

Synthesis must produce circuits that meet the designer's timing constraints

Wire delay based on gathered empirical data from past chips or

estimated by typical (average) wirelength given by Rent's rule – see extra slides at the end of the lecture

- Function of FO
- Function of block size

Lumped RC model used

fanout					
5					
4					
3			cap		
2					
1					
	100	500	1000	2000	3000

block size (gates)

Delay Modeling: Review

Components of cycle time

- Critical path delay, setup time, clock skew, clock-to-Q

Gate delay modeling

- Fast delay evaluation: look up tables
- Conservative models: worst-case assumptions
- Derived by running SPICE simulations of cells in the library

Wire delay modeling

More accurate

- Elmore model, RC tree, AWE
- Statistical wirelength modeling

Typically table look ups – or extracted values are used

Library (.lib) Embodies Gate/Wire Delay Info

Contains for each cell:

- Functional information: cell = a *b * c
- Timing information: function of
  - input slew
  - intrinsic delay
  - output capacitance

  non-linear models used in tabular approach
- Physical footprint (area)
- Power characteristics

Wire-load models - function of

- Block size
- Fan-out

Elements of Timing Verification

To verify circuit timing need

- Accurate delay calculation
- Timing analysis engine

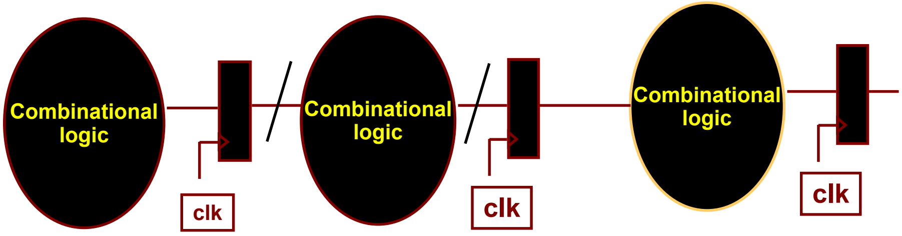
Delay calculation

- Delay numbers for gates
- Delay numbers for wires

Timing analysis engine

- Considering clock network and FF/latches
- Circuit path analysis

Elements of Static Timing Verification - 2

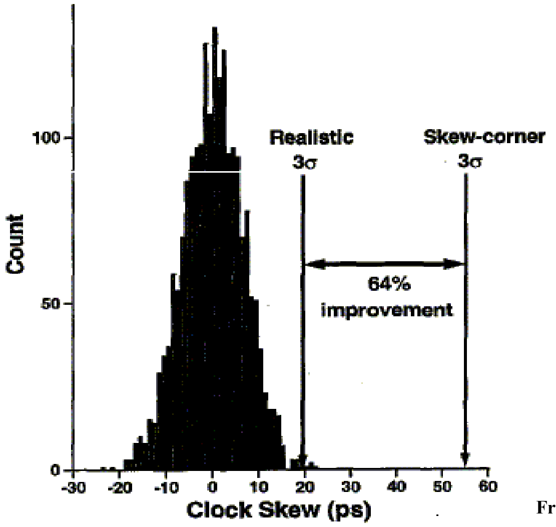


Clocking issues

- Regimes: single-phase, two-phase, multi-phase
- overlapping, non-overlapping
- qualified clocks, clock skew

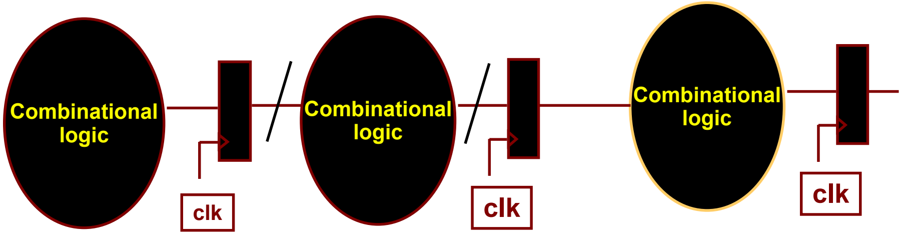
Clock Skew

The clock may be delivered to each FF at a different, unsynchronized, time



From Dennis Sylvester

Elements of Static Timing Verification - 3



Delay calculation procedure

- Longest graphical path
- Longest true delay

Method of calculation

- ``Batch mode''
- Incremental

Typical Simplifications

Clocking issues

- clocking regime treated as orthogonal issue

Delay modeling

- gate - fixed pin-to-pin delays,
- interconnect - non-linear effects captured in tables
- Process-voltage-temperature - worst-case values used

Delay calculation

- ``extract'' combinational logic from sequential circuit
- Choose for accuracy
  - Simple ``longest-path analysis''
  - Boolean analysis excluding false paths

Elements of Timing Verification

To verify circuit timing need

- Accurate delay calculation
- Timing analysis engine

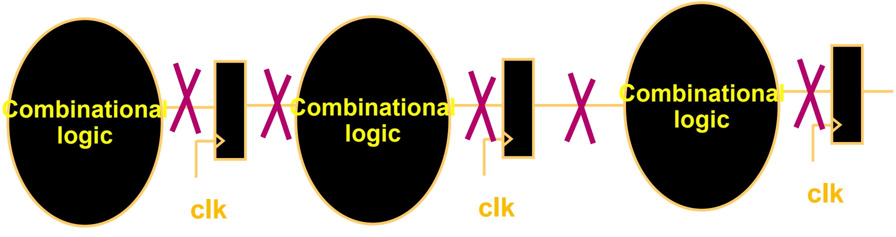
Delay calculation

- Delay numbers for gates
- Delay numbers for wires

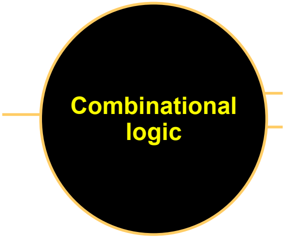
Timing analysis engine

- Considering clock network and FF/latches
- Circuit path analysis
  - Topologically/graphically based
  - Including Boolean/functional pruning

Approach -reduce to combinational



original circuit



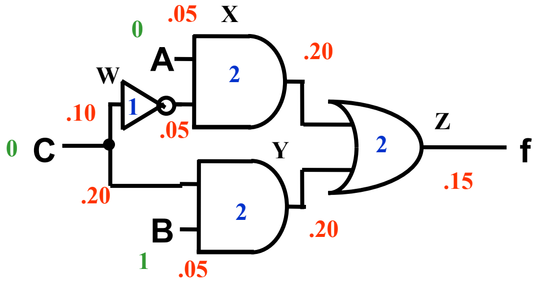
extracted block

Each combinational block

Arrival time in green

Interconnect delay in red

Gate delay in blue



What's the right mathematical object to use to represent this physical object?

Problem formulation - 1

Use a labeled *directed* graph

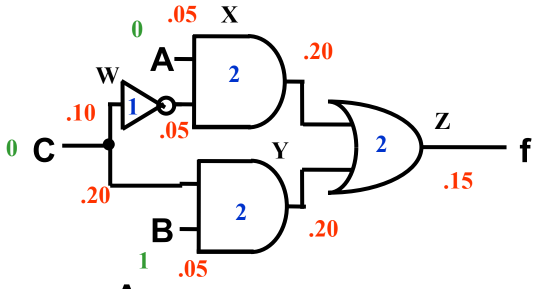
G = <V,E>

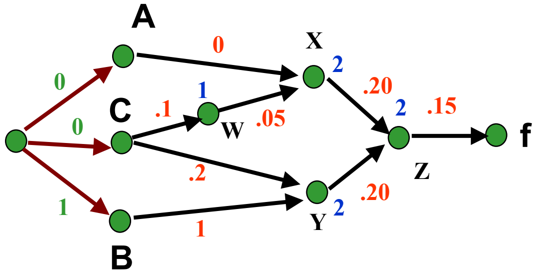
Vertices represent gates, primary inputs and primary outputs

Edges represent wires

Labels represent delays

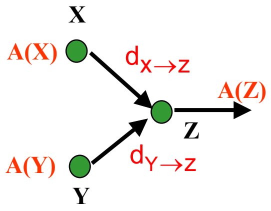
Now what do we do with this?





Problem formulation - Arrival Time

Arrival time A(v) for a node v is time when signal arrives at node v



$$A(\upsilon) = \max_{u \in FI(\upsilon)} (A(u) + d_{u\to\upsilon})$$

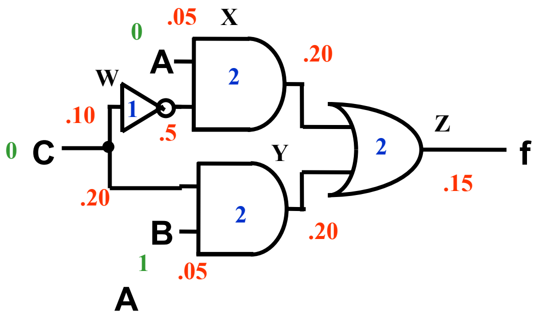
where $d_{\upsilon\to u}$ is delay from υ to u, *FI(υ)* = {X, Y}, and υ = {Z}.

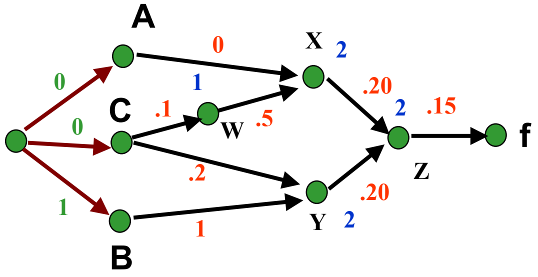
Problem formulation - 2

Use a labeled *directed* graph

G = <V,E>

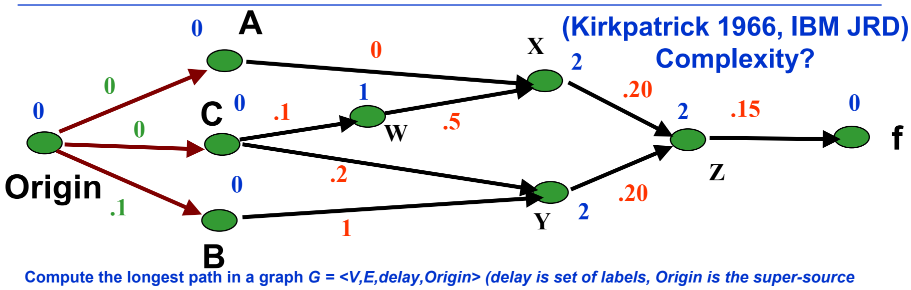
Enumerate all paths - choose the longest?





Problem formulation - 3

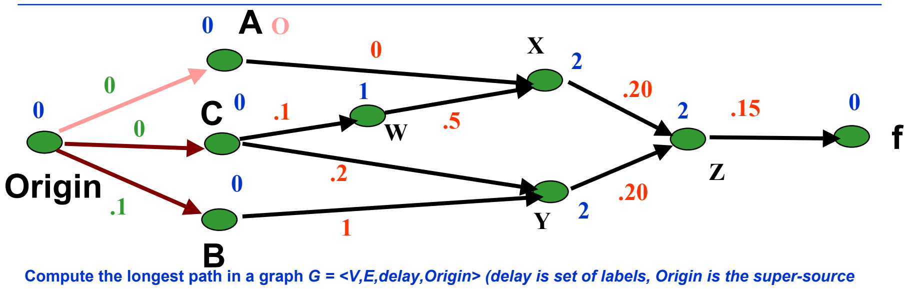
(Kirkpatrick 1966, IBM JRD)
Complexity?



Compute the longest path in a graph *G = <V,E,delay,Origin> (delay is set of labels, Origin is the super-source of the DAG)*

```
Forward-prop(W){
    for each vertex v in W
            for each edge <v,w> from v
                        Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))
                        if all incoming edges of w have been traversed
                                    add w to W
}
Longest path(G)
    Forward_prop(Origin)
}
```

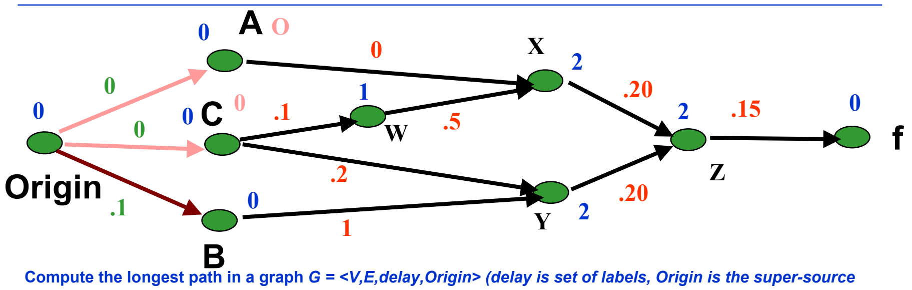
Algorithm Execution



Compute the longest path in a graph *G = <V,E,delay,Origin> (delay is set of labels, Origin is the super-source of the DAG)*

```
Forward-prop(W){
    for each vertex v in W
            for each edge <v,w> from v
                        Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))
                        if all incoming edges of w have been traversed
                                    add w to W
}
Longest path(G)
    Forward_prop(Origin)
}
```

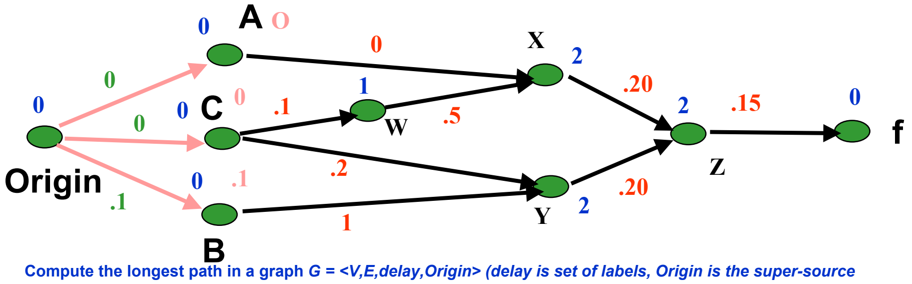
Algorithm Execution



Compute the longest path in a graph *G = <V,E,delay,Origin> (delay is set of labels, Origin is the super-source of the DAG)*

```
Forward-prop(W){
    for each vertex v  in W
        for each edge <v,w> from v
            Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))
            if all incoming edges of w have been traversed
                add w to W
}
Longest path(G)
    Forward_prop(Origin)
}
```

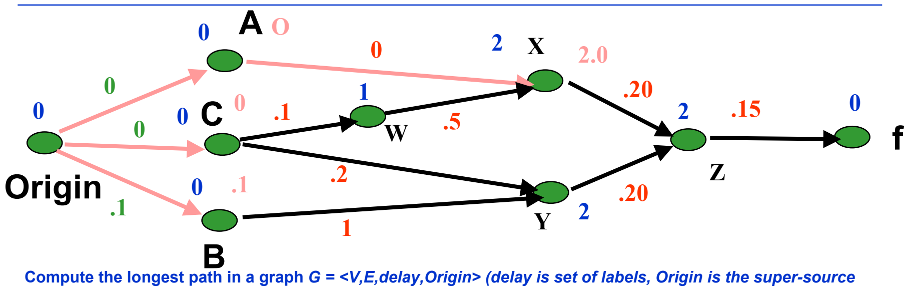
Algorithm Execution



Compute the longest path in a graph *G = <V,E,delay,Origin> (delay is set of labels, Origin is the super-source of the DAG)*

```
Forward-prop(W){
    for each vertex v  in W
        for each edge <v,w> from v
            Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))
            if all incoming edges of w have been traversed
                add w to W
}
Longest path(G)
    Forward_prop(Origin)
}
```

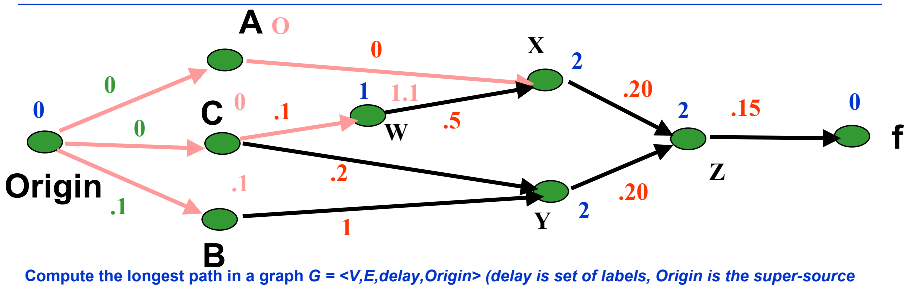
Algorithm Execution



Compute the longest path in a graph *G = <V,E,delay,Origin> (delay is set of labels, Origin is the super-source of the DAG)*

```
Forward-prop(W){
    for each vertex v  in W
        for each edge <v,w> from v
            Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))
            if all incoming edges of w have been traversed
                add w to W
}
Longest path(G)
    Forward_prop(Origin)
}
```

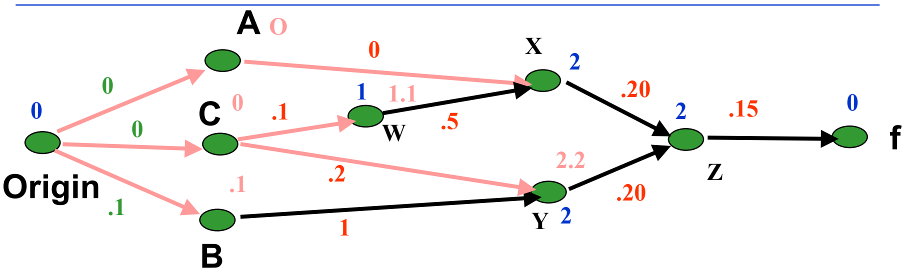
Algorithm Execution



Compute the longest path in a graph *G = <V,E,delay,Origin> (delay is set of labels, Origin is the super-source of the DAG)*

```
Forward-prop(W){
    for each vertex v  in W
        for each edge <v,w> from v
            Final-delay(w) = max(Final-delay(w), delay(v) + delay (w) + delay(<v,w>))
            if all incoming edges of w have been traversed
                add w to W
}
Longest path(G)
    Forward_prop(Origin)
}
```

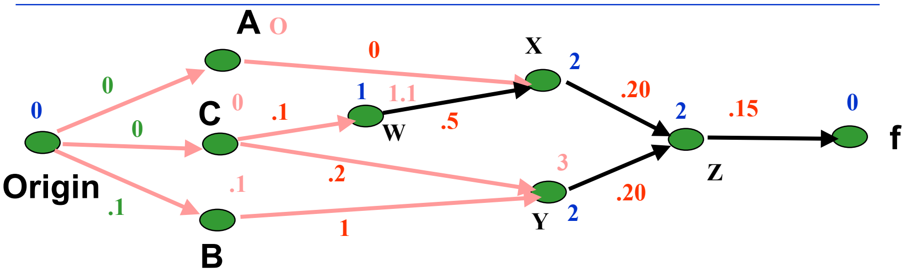
Algorithm Execution



Compute the longest path in a graph *G = <V,E,delay,Origin> (delay is set of labels, Origin is the super-source of the DAG)*

```
Forward-prop(W){
    for each vertex v in W
        for each edge <v,w> from v
            Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))
            if all incoming edges of w have been traversed
                add w to W
}
Longest path(G)
    Forward_prop(Origin)
}
```

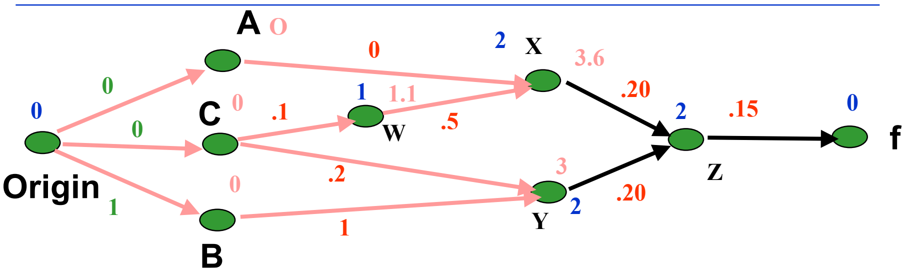
Algorithm Execution



Compute the longest path in a graph *G = <V,E,delay,Origin> (delay is set of labels, Origin is the super-source of the DAG)*

```
Forward-prop(W){
    for each vertex v in W
        for each edge <v,w> from v
            Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))
            if all incoming edges of w have been traversed
                add w to W
}
Longest path(G)
    Forward_prop(Origin)
}
```

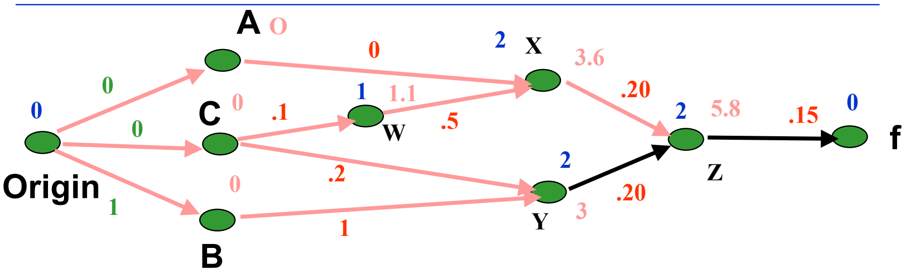
Algorithm Execution



Compute the longest path in a graph *G = <V,E,delay,Origin> (delay is set of labels, Origin is the super-source of the DAG)*

```
Forward-prop(W){
    for each vertex v  in W
            for each edge <v,w> from v
                        Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))
                        if all incoming edges of w have been traversed
                                    add w to W
}
Longest path(G)
    Forward_prop(Origin)
}
```

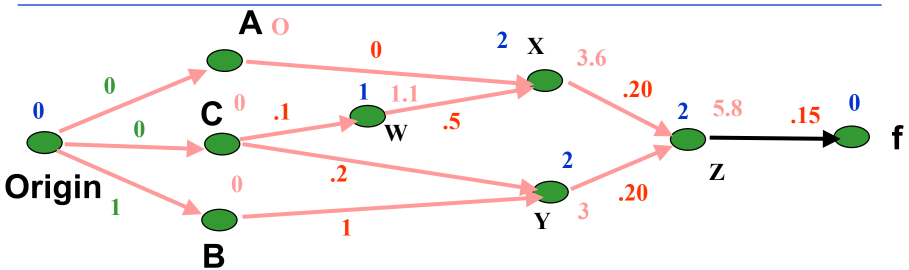
Algorithm Execution



Compute the longest path in a graph *G = <V,E,delay,Origin> (delay is set of labels, Origin is the super-source of the DAG)*

```
Forward-prop(W){
    for each vertex v  in W
            for each edge <v,w> from v
                        Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))
                        if all incoming edges of w have been traversed
                                    add w to W
}
Longest path(G)
    Forward_prop(Origin)
}
```

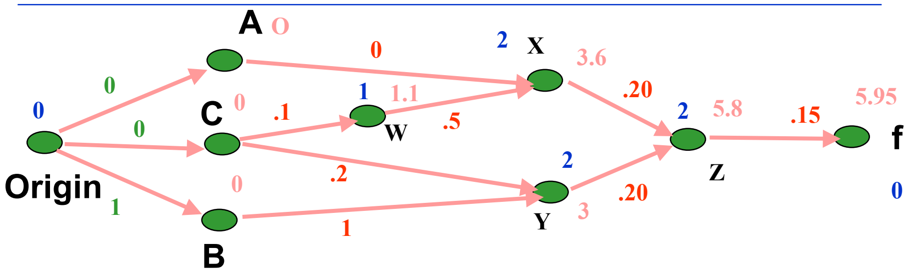
Algorithm Execution



Compute the longest path in a graph *G = <V,E,delay,Origin> (delay is set of labels, Origin is the super-source of the DAG)*

```
Forward-prop(W){
    for each vertex v in W
        for each edge <v,w> from v
            Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))
            if all incoming edges of w have been traversed
                add w to W
}
Longest path(G)
    Forward_prop(Origin)
}
```

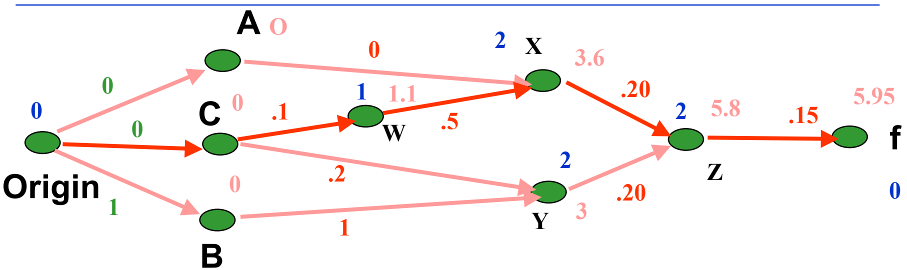
Algorithm Execution



Compute the longest path in a graph *G = <V,E,delay,Origin> (delay is set of labels, Origin is the super-source of the DAG)*

```
Forward-prop(W){
    for each vertex v in W
        for each edge <v,w> from v
            Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))
            if all incoming edges of w have been traversed
                add w to W
}
Longest path(G)
    Forward_prop(Origin)
}
```

Critical Path (sub-graph)



Compute the longest path in a graph G = *<V,E,delay,Origin> (delay is set of labels, Origin is the super-source of the DAG)*

```
Forward-prop(W){
      for each vertex v  in W
                  for each edge <v,w> from v
                                    Final-delay(w) = max(Final-delay(w), delay(v) + delay(w) + delay(<v,w>))
                                    if all incoming edges of w have been traversed
                                                      add w to W
}
Longest path(G)
      Forward_prop(Origin)
}
```

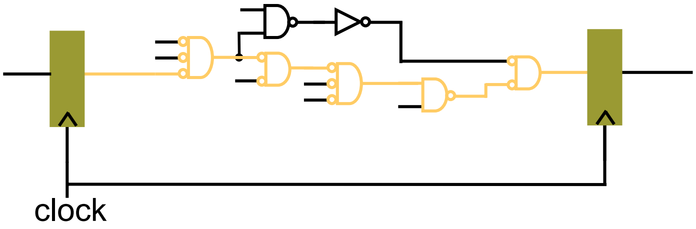
Timing for Optimization: Extra Requirements

Longest-path algorithm computes arrival times at each node

If we have timing constraints, need to propagate slack to each *node*

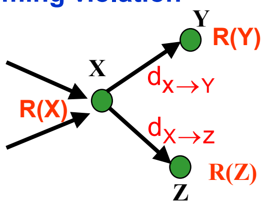
- A measure of how much timing margin exists at each node
- Can optimize a particular branch

Can trade slack for power, area, robustness



Required Time

Required time R(v) is the time before which a signal must arrive to avoid a timing violation



Required time is user defined at output:

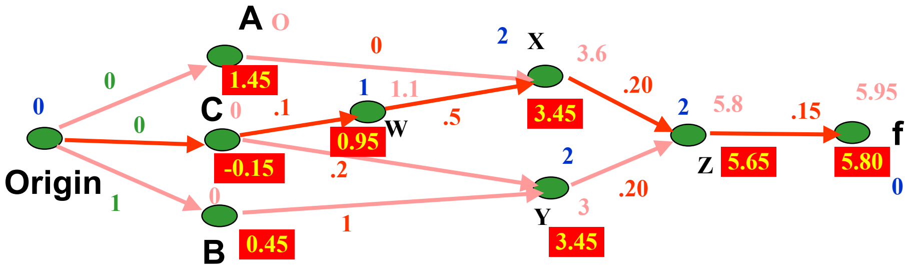
$$R(v) = T - T_{setup}$$

Then recursively

$$R(\upsilon) = \min_{u \in FO(\upsilon)} (R(u) - d_{\upsilon \to u})$$

where $FO(\upsilon) = \{Y, Z\}$ and $\upsilon = \{X\}$

Required Time Propagation: Example



Assume required time at output R(f) = 5.80

Propagate required times backwards

Timing Slack

From arrival and required time can compute slack. For each node v:

$$S(\upsilon) = R(\upsilon) - A(\upsilon)$$

Slack reflects criticality of a node

Positive slack

- Node is not on critical path. Timing constraints met.

Zero slack

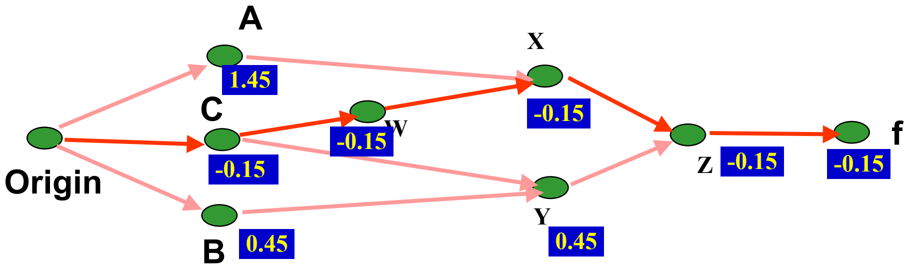
- Node is on critical path. Timing constraints are barely met.

Negative slack

- There is a timing violation

Slack distribution is key for timing optimization!

Timing Slack Computation: Example



Compute slack at each node

$$S(\upsilon) = R(\upsilon) - A(\upsilon)$$

Timing Slack Properties

R. Nair et al, “Generation of Performance Constraints for Layout”, TCAD 1989.

Path through a timing graph: $\rho = < v_1, v_2, \ldots, v_k >$

Path slack: $S(\rho) = R(v_k) - A(v_1) - \sum_{i=1}^{k-1} d_{v_i \to v_{i+1}}$

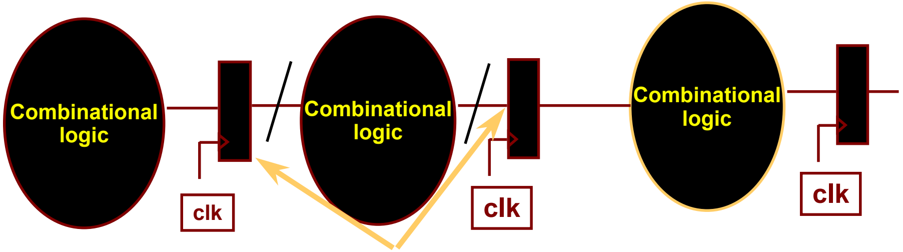
Path slack reflects slack at output assuming no other path in circuit is active

Lemma 1: For any path $\rho = < v_1, v_2, \ldots, v_k >$ **through circuit,** $S(v_i) \le S(\rho)$, **for** $1 \le i \le k$

Lemma 2: For each cell v $\in$ **V there is some path** ρ_v **such that** $S(v) = S(\rho_v)$

By increasing delay on only one cell in the path, can set path slack to zero -> other cells also have zero slack.

Approach of Static Timing Verification



- computing longest path delay
  - full path enumeration - potentially exponential
  - longest path algorithm on DAG (Kirkpatrick 1966, IBM JRD) (*O(v+e)* or *O(g + p)*)
- Currently in successful application on even the largest (>10M gate) circuits
- has two challenges:
  - asynchronous sub-circuits - limited gate-level simulation
  - false paths - ubiquitous and problematic

Elements of Timing Verification

To verify circuit timing need

- Accurate delay calculation
- Timing analysis engine

Delay calculation

- Delay numbers for gates
- Delay numbers for wires

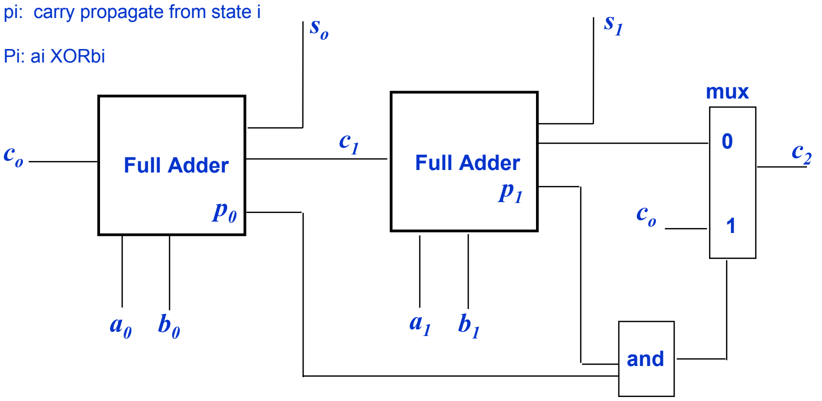
Timing analysis engine

- Considering clock network and FF/latches
- Circuit path analysis
    - Topologically/graphically based
    - Including Boolean/functional pruning

Interesting Example: Carry Bypass Adder

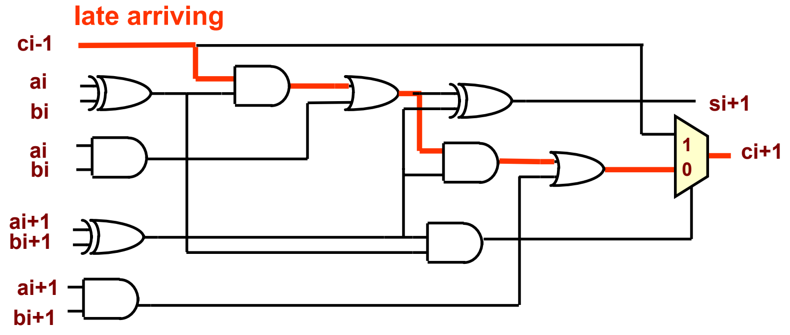
pi: carry propagate from state i

Pi: ai XORbi



Lehman, Birla - IRETrans. Electron. Comput. , 1961
V. Oklobdzija - Jrnl. of VLSI Signal Processing, 1991

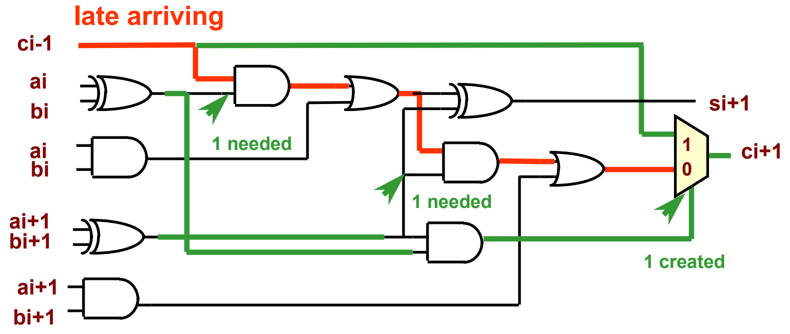
Inside Carry Bypass Adder - 1



Longest graphical/topological path runs along carry chain from stage to stage

Longest path analysis would identify red path as critical

Inside Carry Bypass Adder - 2



To sensitize red path we need: ai⊕bi && ai+1⊕ bi+1

But: red path is false because when this condition is true MUX selects “1” input, i.e. directly from ci-1

Instead shorter green paths are sensitized and red path is not the critical path of the circuit

False paths first observed by V. Hrapcenko (Soviet Math. Dokl. 1978)

Capturing Functional Behavior in Analysis

Failure to remove false paths leads to conservative delay estimation

Looking for a path delay calculation approach that :

- Is conservative
- Is not (overly) pessimistic
- Incorporates functional (Boolean) behavior
    - Eliminates false paths from analysis

Two approaches to false path elimination

- User-specified path exceptions (still mainstream)
- Automatic false-path detection

False Path Detection

Sensitization criterion: a condition under which a signal transition at gate input will propagate to the gate output

Two-vector (transition mode) condition

- More complex: two-vector condition with (2^n * 2^n) vs. single vector condition (2^n space)
- Does not satisfy monotone speedup property
    - Path delay increased when gate delay decreases

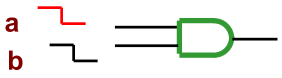
One-vector (floating mode) condition

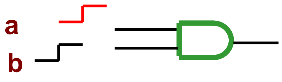
- Previous node value is (conservatively) indeterminate

Chen-Du Path Sensitization Condition

Path from a gate input to the gate output is true iff the input gives

1. The earliest controlling value
2. The latest non-controlling value if all inputs are non-controlling





Functional timing analysis

- Test each path for falsity
- Implicit path delay sensitization (PODEM)

Enhancements of STA – project ideas

Incremental timing

Incorporation of deep sub-micron effects - crosstalk

Incorporation of physical variation → timing variation – statistical timing

Current Status of Static Timing Verification

Static timing verification is now the principal method for verifying timing in the majority of digital circuits

- ASICs:
  - Gate-level models from Semiconductor vendors
  - Driven by fast interpolation from tables
  - Estimated (wire-load model) or back-annotated capacitances
- Custom (e.g. Intel microprocessor)/COT (e.g. nVidia, ATI graphics chips)
  - Mixture of above and transistor-level static-timing verification
  - Transistor level based on extracted device and wiring attributes
  - Simulation of each transistor network builds accurate model

Extras

Rent’s rule

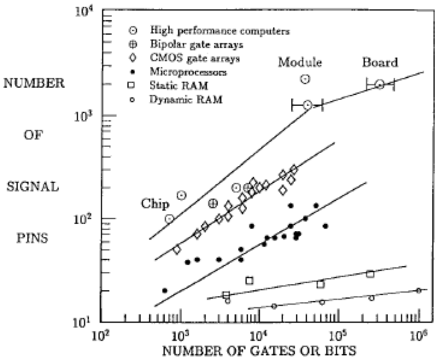
RC Models

Statistical Wire Length Estimation

Before P&R, need to predict wire length for each net

Can predict average (typical) wirelength based on empirical statistical regularities

Early on data showed that there exist strong correlation between number of IO terminals and number of gates in the block



(Bakoglu)

Rent’s Rule

In 1960, E.F.Rent showed that

$$N_p = K(N_g)^{\beta}$$

where N_p =no. of external signal connections on a block, N_g = no. of logic gates in a block, K accounts for # of pins per gate, and β is Rent’ s constant – is a measure of many of the gates in a circuit block need to communicate with the outside world.

Rent’ s Rule is very circuit-fabric (architecture) specific

Chip Type	Rent’s constant	Proportionality Constant K
Static Memory	0.12	6
Microprocessor	0.45	0.82
Gate Array	0.5	1.9

Typical values of Rent’s Coefficient (Bakoglu)

Statistical Wirelength Estimation

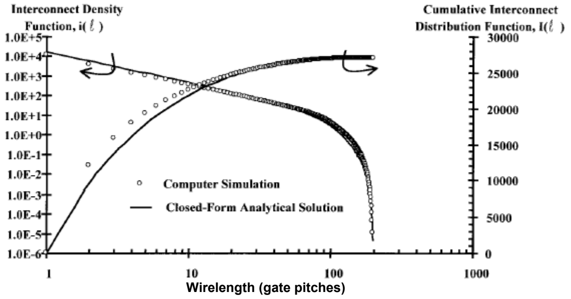
Estimate average wiring length for point-to-point nets by applying Rent's rule recursively:

- Partition the chip into hierarchical divisions
- Estimate the connections between partitions by Rent' s Rule

$$L_{avg} = f(N_g, \beta)$$

Wire length also depends on fan out of the net

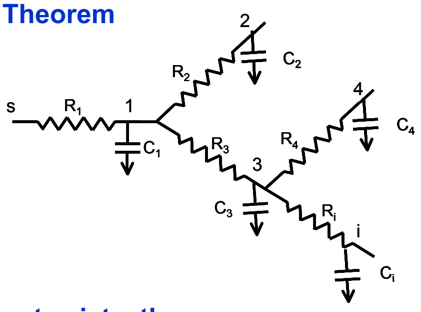
$$L_{avg}(FO) = L_{avg}(1 + 0.4(FO - 1))$$



(J. Davis)

Wire Delay Modeling: RC Trees

For RC trees (branching networks), delay can be estimated using Rubinstein-Penfield Theorem



If a single dominant time-constant exists, then

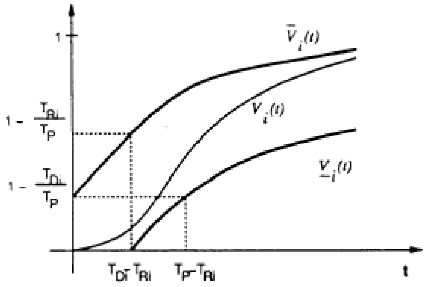
$$\tau_i \propto \sum_{k=1}^{N} C_k R_{i,k}$$

where $R_{i,k} = \sum R_j \Rightarrow (R_j \in [path(i \rightarrow s) \cap path(k \rightarrow s)])$

E.g. $\tau_i == C_1R_1 + C_2R_1 + C_3(R_1 + R_3) + C_4(R_1 + R_3) + C_i(R_1 + R_3 + R_i)$

Wire Delay Modeling: RC Trees

The theorem also establishes precise bounds on voltage waveforms



Same limitations as for Elmore constant

- RC- tree topologies (not applicable with inter-nodal capacitances)
- Instantaneous input transition times